NBSIR 76-1033 (R)

Minicomputers: An Attitude

T. G. Lewis

University of Southwestern Louisiana
Lafayette, Louisiana

March 1976

Final Report

Prepared for
The NBS/NSF Software Engineering Handbook
Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D. C. 20234

NBSIR 76-1033

MINICOMPUTERS: AN ATTITUDE

T. G. Lewis

University of Southwestern Louisiana
Lafayette, Louisiana

March 1976

Final Report

Prepared for
The NBS/NSF Software Engineering Handbook
Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D. C. 20234

U.S. DEPARTMENT OF COMMERCE, Elliot L. Richardson, *Secretary*
James A. Baker, III, *Under Secretary*
Dr. Betsy Ancker-Johnson, *Assistant Secretary for Science and Technology*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Acting Director*

TABLE OF CONTENTS

		Page
1.	Introduction	1
	1.1 Size	2
	1.2 Speed	3
	1.3 Cost	4
	1.4 Applications	4
	1.5 Peripherals	5
	1.6 Software	5
	1.7 Design	6
2.	Architecture of Minicomputers	6
	2.1 Microprogrammed Versus Random Logic For Minis	7
	2.2 Central Bus Versus Distributed Bus	10
	2.3 Special Register Versus General Register	12
	2.4 Instruction Architecture	13
	2.5 Summary	18
3.	Minicomputer Programming	20
	3.1 Case Study I: Mini A (Simple Machine)	20
	3.2 Example A.1	23
	3.3 Example A.2	24
	3.4 Example A.3	24
	3.5 Special Techniques	25
	3.6 Case Study II: Mini B	25
	3.7 Example B.1	30
	3.8 Special Techniques	32
	3.9 Example B.2	32
	3.10 Case Study III: Mini C	32
4.	Counterpoint 1975	42
	4.1 The Multilevel Mini	42
	4.2 Summary	44
5.	References	44

LIST OF FIGURES

Page

Figure 1. Cost versus Complexity for Random Logic and M-processor 8

Figure 2. A Typical M-processor (simplified) 9

Figure 3. A Central Bus Organization 11

Figure 4. Word-Paged Format 14

Figure 5. Shift/Rotate Instructions Work On 17-bits 15

Figure 6. I/O Vectors For Automatic I/O 19

Figure 7. Mini A 21

Figure 8. Mini B As the Programmer Sees It 27

Figure 9. The Program Space And Data Space For Mini C Main Memory 34

Figure 10. The Addressing Modes of Mini C 36

Figure 11. A Sample HLL Program For Mini C 38

MINICOMPUTERS: AN ATTITUDE

T. G. LEWIS

Minicomputers are defined in dozens of ways: by word length, memory size, speed, cost, applications, peripherals, software, and design. The definition used here includes all limited resource computers; emphasis is placed on a minicomputer attitude that attaches importance to the design of simple, straightforward, special purpose, dedicated computing systems.

Minicomputer architectures are categorized according to type of control (random logic or microprogrammed), bus structure (distributed or central), number of working registers, and instruction types. Three demonstration minis are used to show how hardware complexity influences software complexity, and consequently, software cost. The comparison suggests that complexity should be forced into hardware, since hardware is less expensive than software. Programming and software emerge as the most significant problems faced within the minicomputer environment. Each minicomputer should support a reasonable high-level language to ease the programming task.

Concluding speculations suggest that minis will overcome current limitations, will incorporate more complexity into hardware, and will use the multi-level nature of software, firmware (microprogrammable elements), and hardware to advantage in special purpose systems.

Key words: Architecture; assembly language; LSI; microprogramming; minicomputer; physical I/O; programming techniques for small computers; stack processing.

1. INTRODUCTION: WHAT IS A MINICOMPUTER? [16]

Traditionally, electronic digital computers have been large, complex, and expensive. In the early years they were incredible devices composed of thousands of vacuum tubes and miles of wiring. Even now, thirty years later, electronic digital computers summon an image of massive hardware and sophisticated software.

In the early sixties, a few small digital computers of limited capacity and power were designed and built. Because of their modest size these machines were not called general purpose digital computers; rather, their low cost and flexibility made them useful as process

controllers or as "programmable hardware" in a variety of applications.

The demand for flexible hardware and the development of special digital techniques (especially for the military) created a market for small computers. This market began to expand rapidly in the late nineteen sixties. Not surprisingly, the small computers were tagged with a common expression: they were "minis".

In the period 1967-1972 many manufacturers entered the minicomputer market. It may also have been during this period of competition and rapid change that computer experts began to feel uneasy about the definition of a minicomputer. Many such definitions have been proposed: a computer on a single circuit board, a computer that occupies less than one cubic foot of space, a 16-bit computer, a computer costing less than twenty thousand dollars, a computer with limited instruction set and small memory. Each is valid only in a limited sense: in terms of storage capacity and instruction set limitations all early computers were minis; a desk calculator qualifies as physically small and inexpensive.

This report will demonstrate that all of the definitions above have something to do with minicomputers. The significant point missed by all of the definitions is that a minicomputer is a state of mind and minicomputing is an attitude.

The minicomputer attitude is concerned with efficient utilization of memory space, optimum use of instructions, low cost, and special purpose computing. It seeks to implement hardware and software systems under extreme constraints. Constraints may include rapid execution speed, reliability under environmental shock, or financial limits.

Before approaching the hardware and software details of minicomputing, let us survey the characteristics of minis: size, speed, cost, application areas, peripherals, software, and design philosophy. In the following, the reader will observe that many of the concepts discussed are also part of "maxi" computing. The purpose of this article is to point out features central to the minicomputer attitude, not to separate mini from maxi. It would indeed be surprising if the two were dramatically different.

1.1 Size

A typical minicomputer occupies about one cubic foot of space, has a maximum capacity of 64K words or bytes, and has 16 bits per word. Very few minicomputers are purchased with memory and cpu, alone. Usually additional backing store of disk, tape, or cassette is purchased along with printers, terminals, and necessary interface and controllers. A complete minicomputer system may require a sizeable amount of physical

space (including a controlled environmental room).

1.2 Speed

Surprisingly, minicomputers are fast. A minicomputer tuned for 32 bit scientific computation is capable of 600nsec. addition (mid-seventies estimate) and 5u sec multiplication [1]. Correspondingly, a typical large scale computer requires 2u sec addition time and 20u sec multiplication time [2]. Hence, in this single simple measure, the minicomputer is two to four times faster than a large computer. (The reader should note the bias purposely introduced by the author.)

Minicomputers are often faster than large computers because they take advantage of technology almost immediately. Using the new generation announcements by large computer manufacturers as a measure of technological change, the following table shows a six year lifetime per generation:

generation	announcement date
1	1952
2	1958
3	1964
4	1970
5	1976

The life time of a large model computer is roughly proportional to the manufacturer's investment. A big investment in technology is necessary to develop, program, and market a large machine and its attendant software. Software, of course, involves considerable cost. Minicomputers, on the other hand, represent to the manufacturer only relatively small investments for hardware. Some of the reasons are listed below:

1. Minis are often sold in quantities to original equipment manufacturers (OEM) and the profit is mainly on the actual hardware (no software is included in the sales).

2. Minis are modular and uncomplicated thus making them adaptable to new technology.

3. Mini customers (OEM people) are usually more sophisticated than the general buyer and therefore rely less upon the manufacturer's service (maintenance, software libraries, etc.).

4. Most mini manufacturers are merely component assemblers and do not develop the technology directly, but instead, buy it from outside suppliers.

As an example of item 2 above, minis were the first computers to employ semiconductor main memories having an order of magnitude greater speed than core memory. In general, a production change

on a mini assembly line effects fewer than one hundred skilled workers while a change in a large main frame manufacturer affects thousands. This enables minicomputer manufacturers to employ a technological advance in a matter of months after it is available.

1.3 Cost

Word size is the most limiting factor in designing low cost computers. This is reflected in the short word length of minis. In the early sixties a minicomputer had 12-bits per word. In the late sixties, and in the seventies a 16-bit word is typical. Recently, several 32-bit minis have been introduced, perhaps indicating a trend toward larger words.

We can classify computers according to word size and cost, at the risk of being rapidly outdated, as follows:

Classification (name)	Word Size (bits)	Cost (thousands $)
Micro	4 to 8	0.1 to 3
Mini	8 to 16	3 to 20
Midi	16 to 24	20 to 100
Large	24 to 32	100 to 1,000
Super	32 to 64	over 1,000

1.4 Applications

The low cost of minis opens the door for a wide variety of applications. Although large computers are employed heavily in military, financial, and corporate applications, very few of these expensive computers are employed in areas dominated by minis. The spectrum of minis' applications is given below:

A. Preprocessing
   Displays, peripherals, buffering.

B. Communications
   Message switching, telemetry, data concentration.

C. Scientific Computing
   Hand/pocket calculators, special purpose equation solvers.

D. Process Control
   Machine tools, production lines, monitoring, laboratory automation.

E. Business Data Processing
   Special purpose inventory or financial control.

F. Command and Control
Guidance and tracking, navigation, cryptography.

Minis are used as part of intelligent terminals, controllers for peripheral storage devices, and front-end or back-end processors. They are low cost alternatives to hardwired data concentrators, or message switchers. In factories they may be disguised as automatic production control machines or as laboratory equipment.

Although minis have traditionally been weak in numerical calculations, they offer definite cost benefit advantages in special purpose scientific and business applications. Packages such as ECAP (Electronic Circuit Analysis Program) have been implemented on minis. "Turnkey" systems, i.e. ready-to-use hardware and software tailored to a special application, are currently available that incorporate a mini as part of a database system for small businesses.

Finally, due to their small size and low cost, minis are used in aerospace applications. These applications commonly require uncomplicated but high speed processing.

1.5 Peripherals

While the cost of a minicomputer cpu is dwindling, the cost of peripherals is still sizeable. Indeed minicomputer peripherals account for a large share of the manufacturer's profits. In 1969 the ratio of total system cost to the minicomputer cpu was 2:1. By the early seventies this ratio had risen to 3:1, and in the mid-seventies it is 4:1.

Disk and tape drives cost as much as a mini cpu (1975 prices). Addition of paper tape I/O devices, cassette, flexible disks, and a line printer drives the cost of a complete system upward. The cost of peripherals will become the major hardware obstacle to penetrating low cost, high volume markets, e.g. appliances, automobiles, home entertainment, industrial control, communications, and education.

In addition to the cost of peripherals, the cost of controllers and interfaces has added to the relative high cost of total systems. The simplest serial device (such as a console teleprinter) may conform to industry standards but still require an additional interface. For example, a teleprinter interface may specify an EIA RS 232-A standard but fail to indicate strapping options or the transmission speed. Therefore, an additional interface device is needed to make the peripheral work harmoniously with the cpu. Only recently have mini manufacturers recognized this problem and begun to produce general purpose I/O ports.

1.6 Software

The software available for a typical mini is limited. Operating

systems are unsophisticated and usually special purpose. Language processors are nearly non-existent except for assembler translation and BASIC. Often translators run on large machines that produce object code for the mini. These are called cross-translators. However, a few minis have FORTRAN, ALGOL, RPG, or COBOL. The trend of the seventies is toward more high level languages [4, 7, 12, 13].

1.7 Design

In the remainder of this decade minicomputers will use more LSI (large scale integration) components to shrink size and electrical power requirements and to provide an economical way of building more sophisticated central processors, controllers, and memories.

LSI employs solid state circuitry and miniaturization to combine thousands of components into a single package collectively called a "chip" (roughly 1" to 2" long, ¼" to 3/4" wide and ¼" thick). The chip is manufactured by the thousands at very low cost, but once the circuits are designed they may not be altered without high cost. Therefore, the LSI chip is inflexible but low cost.

A minicomputer must remain flexible and extensible to be marketable. This seemingly causes a conflict between LSI technology and minicomputer design. The conflict is resolved in part by the acceptance of a limited variety of minicomputer organizations (inflexibility), modularity (extensibility), and microprogramming (an intermediate level of flexibility imposed between the hardware and software, e.g. firmware).

In the following section we discuss the architecture of minicomputers. Keep in mind the forces of cost reduction and the counterforces of flexibility. Minicomputer designers use central bus design, microprogramming, and modularity to balance cost and flexibility. Ultimately designer decisions reflect in the cost and complexity of software, and as we have seen, software is the major facet of the minicomputing attitude.

2. ARCHITECTURE OF MINICOMPUTERS

Minicomputer designs vary tremendously from model to model. As a result, it is difficult to compare the performance of model x to that of model y even within a single manufacturer. Instead, we will examine the dominant types of organization as follows.

1. MICROPROGRAMMED VERSUS RANDOM LOGIC

2. CENTRAL BUS VERSUS DISTRIBUTED BUS

3. SPECIAL REGISTER VERSUS GENERAL PURPOSE REGISTER VERSUS STACK

4. INSTRUCTION ARCHITECTURE

Items 1-4 are design features which also can apply to larger machines.

2.1 Microprogrammed Versus Random Logic For Minis

A random logic architecture implies that the minicomputer's instruction set is hardwired. The only way to modify the effects of an instruction is to alter the circuitry. Such a system can be thought to be a two-level system: the hardware and the software.

A microprogrammable computer, on the other hand, is a three level computer: hardwired logic (perhaps LSI chips), firmware logic (a memory containing microinstructions which control the cpu), and the software logic (a memory containing instructions which are interpreted by sequences of microinstructions). Each software instruction taken from main memory is carried out by a sequence of firmware microinstructions which in turn control the sequencing of hardware functions. The firmware routines stored in the control memory constitute a simulator or interpreter called an emulator. In other words, the software instructions are emulated by a microprogram which runs on a microprocessor (m-processor),

The incremental cost of increased complexity in an m-processor is based on the cost of high speed control memory (see Figure 1). Control memory is a (relatively) small read-only-store that contains m-processor control words or microinstructions. As the size of the microprogram increases it is necessary to add pages (blocks or "chunks") of control memory. The step function of Figure 1 represents the incremental cost in expanding the instruction set of an m-processor.

Figure 2 shows how the control memory fits into the overall system. During execution of instructions from main memory (i.e. interpretation on the m-processor), the control memory acts as a ROM (read-only-memory). When we wish to alter the contents of control memory (to load a different interpreter for machine instruction(s)) the ROM becomes a WCS (writeable control store). The WCS appears to be an output device during a write cycle. From the processor's view, the whole reload had better be one cycle, since after initiation of the transfer no processor intervention can occur. Transfer failure or error can easily leave the processor inoperable until manual intervention.

Actually the WCS/ROM control memory may be part of main memory. Microinstructions are fetched from one segment of main memory and machine instructions fetched from another segment. If we think of microprogramming in this context, then emulation--simulation of one machine's instruction set on another machine--is simply another level of interpretation below machine language.

The * in Figure 1 marks the cost-effective cross-over from random logic to m-processor design. It is difficult to estimate the exact location of the * which we will call the "hardware shift" point.

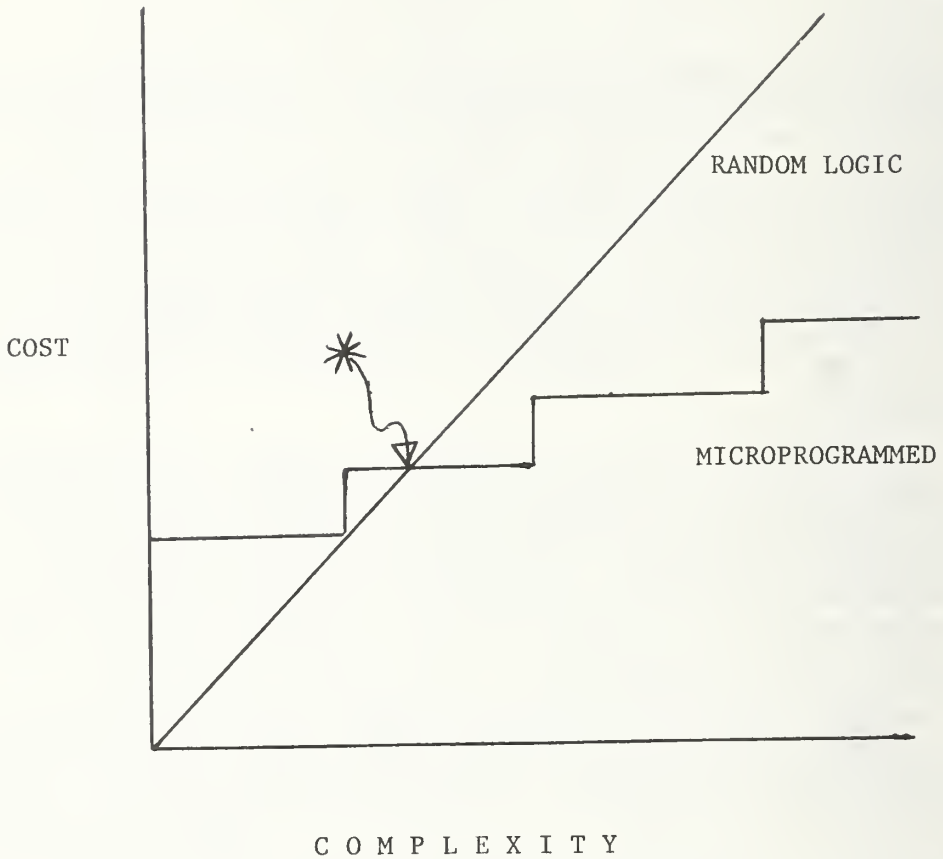


Figure 1. Cost versus Complexity for Random Logic and M-processor

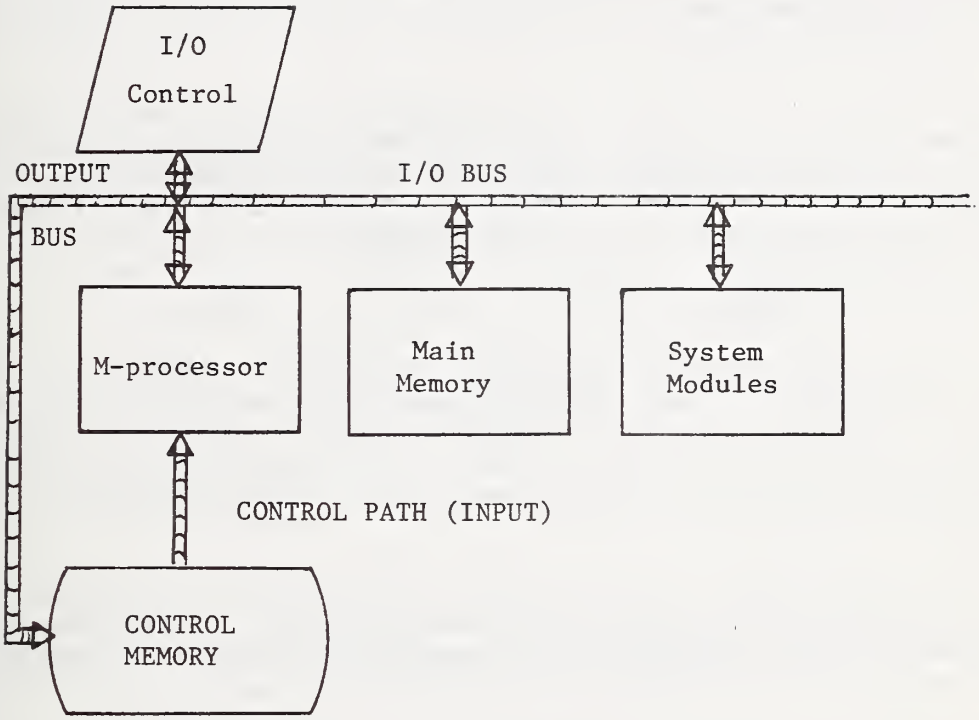


Figure 2. A Typical M-processor (simplified)

The hardware shift point moves because of two factors: decreasing cost of random logic through LSI technology, and decreasing cost of control memory through memory technology (which includes LSI advances).

Most minicomputers are microprogrammed but only a few are microprogrammed by the user. User microprogramming provides a way to tailor the mini to individual applications. There are, however, pitfalls that await the uninformed who do not understand the complexities and limitations of microprogramming.

Manufacturers of non-user microprogrammed minis usually do not advertise their mini as being microprogrammed. Control memories of such machines are unalterable and implemented in FROM (fusable-link), PROM (programmed), or some form of ROM. We will treat this type of mini logically the same as we treat random logic minis.

2.2 Central Bus Versus Distributed Bus

One of the most important design criteria for manufacturers of minis is modularity. Since the applications and requirements for hardware vary radically from system configuration to configuration some means of modularity is highly desirable. For example, when a variety of I/O devices are to be attached how can the designer provide a single universal interface?

The central bus, or universal bus structure was invented to alleviate the problem of interfacing and modularity. Surprisingly, the universal bus impacts heavily upon programming because it eliminates the necessity for I/O instructions (see next section).

Figure 3 shows how we might view a universal-bus mini. Each system module is physically attached to the bus and assigned a logical location called its bus address. The bus address is used to access data regardless of its origin. For instance, the status register of a peripheral device is treated the same as a main memory word by the programmer.+

Early minicomputers were organized around a collection of internal buses. Such a distributed bus allows simultaneous transfer of data

---

+MASTER/SLAVE relation is one between two modules on the bus which are involved in a transter. The MASTER assumes control of the bus. The priority system on typical minis has two facets:

Bus grant: priority determines which system module gets the bus. Once granted, the module cannot be preempted.

Cpu grant: interrupts from the bus (generated by activities of system modules) are processed by the cpu at the priority of the bus grant. The cpu will be preempted by requests of strictly higher priority.

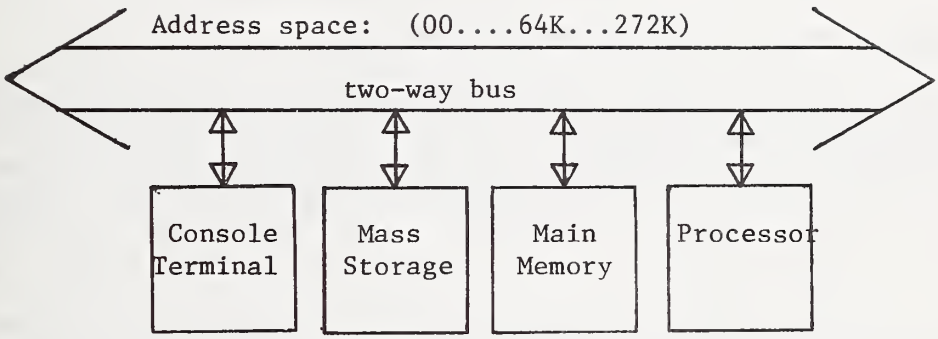


Figure 3. A central bus organization. Each system module is given one or more addresses on the bus.

between system components, but decreases hardware flexibility. The universal bus, on the other hand, increases modularity, but often contributes to congestion.

A priority system is established along the bus to determine which system module is MASTER and which one is SLAVE during a bus transfer. Interrupts are allowed only by modules of equal or higher priority.

Various manufacturers have trademark names for these devices which are actually "intelligent buses" because they perform a limited amount of processing, e.g. shifting, timing.

2.3 Special Register Versus General Register

The execution speed, amount of program code, and low level programming ease of any computer is closely related to the number of working registers made available to the assembler programmer or translator writer. Traditionally, an accumulator, multiplier/quotient, and possibly an index register were all that a programmer could expect. Such minis are called register - register minis with special purpose registers.

Decrease in hardware costs have brought about an orientation to multiple registers. In a machine with, say eight registers, R0, R1, ... R7, the function of the registers will usually be general purpose. This increases the power of mini-computing by decreasing the number of load/store operations.

Recently minicomputers have been designed to support high level system implementation languages [4,7] . Such language-directed minis are based on a single pushdown stack of registers rather than a few working registers. Thus they are in a sense unlimited-register machines having the ability to access an unlimited stack of registers.

Another possibility for register organization exists. Suppose we bypass registers entirely and allow the machine-level instructions to modify the contents of main memory directly? Two and three-address machines may, for example, allow storage-storage operations directly without movement of data to working registers. The central bus structure eliminates the need for direct use of working registers because the working register and main memory registers are both attached to the bus as system modules. In actual operation, however, the operands from main memory may be held temporarily in internal registers without the programmer's knowledge. To the programmer it appears that it is possible to write programs without ever referencing a working register (see the section on Mini B).

In the following section we will discuss three types of minicomputers differing in the number and purpose of their working registers. The machines will be (A) a single address, (B) a two-address, and (C) a

stack, or zero address minicomputer. Before doing so, however, let us examine the instruction architecture of a typical minicomputer.

2.4 Instruction Architecture

The most notable feature of a minicomputer's instruction set is its limitation. Often multiply/divide instructions are an extra cost option and floating-point arithmetic an after-thought. Arithmetic addition and subtraction is standard but the compare and logical instructions are typically limited. For example, the exclusive-or instruction may be lacking and instead must be simulated with two AND instructions and one OR instruction.

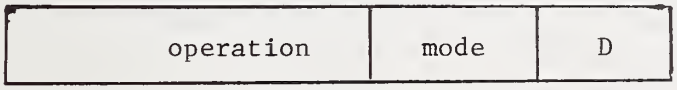
Branching and subroutine calls may be sophisticated for small machines. The design of subroutine linkage control can be elegant for minis that employ a pushdown stack mechanism. Often, however, recursive calls to subroutines are difficult because return addresses are saved in the first word of the called subroutine or in a working register.

Machine/assembly code is sometimes a difficult notation for programming or to read later because of the way instrucitons are packed into an instruction word. Regretfully, few minicomputer designers attempt to ease the programmer's job through careful design of instruction sets and mnemonics. This trend may be reversed in the future due to the high cost of software development.

Minicomputers are well known for their special purpose instructions. A mini used for communications control most likely will have a "generate ASCII parity" instruction that produces a parity bit. Polling is frequently done by a mini. In this situation a single instruction is used to determine which bit is set in a word (scanning left-to-right).

The main problem with minicomputer instruction sets arises from the short instruction word. A 16-bit mini must incorporate a variety of addressing modes in order to address reasonably large memories and at the same time represent a reasonably large instruction set. In addition, they must be able to process byte-length data in addition to word-length data. This places severe demands upon the 16-bit word length.

The effective address of an operand is calculated in six or seven basic ways. The simplest mode of addressing is to use a part of the instruction word, say D, as the absolute address of an operand.



instruction word

Naturally the magnitude of D limits the instruction to only a

few hundred bytes of "addressability". If memory is segmented into blocks called pages, then a paged address is computed by adding the page number to the displacement value, D.

Figure 4. Word-Paged format.

Paged addressing is inconvenient and makes the problem of relocating programs to new pages unnecessarily complicated. If the program counter or a special base register is used in combination with the displacement value D, then a relative address may be computed.

Since arrays and tables are frequently searched by minicomputers, a convenient addressing mode is the indexed mode in which the contents of an index register are added to D.

Further, since searching is done by sequentially examining contiguous locations the auto step addressing mode is helpful. A main memory word or register is used as an index or as an address of the operand. The register or memory word is incremented/decremented automatically, either before (pre-index) or after being used (post-index).

Assume D is the contents of a field within an instruction word and M is the mode of addressing. The effective address EA is given for various values of M, below.

M = 0	ABSOLUTE	:	EA = D
1	PAGED	:	EA = page number + D
2	REGISTER	:	EA = a register number
3	PROGRAM RELATIVE	:	EA = contents of program counter + D
	BASE RELATIVE	:	EA = contents of base register + D
4	INDEXED	:	EA = D + contents of index register
5	INDIRECT	:	EA = contents of address specified by some mode
6	AUTOSTEP	:	After the EA is computed, increment or decrement the indirect address used to fetch the data

The shift/rotate instructions on a mini often depart from those of a large machine since only a basic set of shift instructions is usually provided. The full set is built-up from the basic set. Often shift length is limited to one bit and multiple-register shifts are made possible with the assistance of an additional carry bit or overflow. See figure 5.

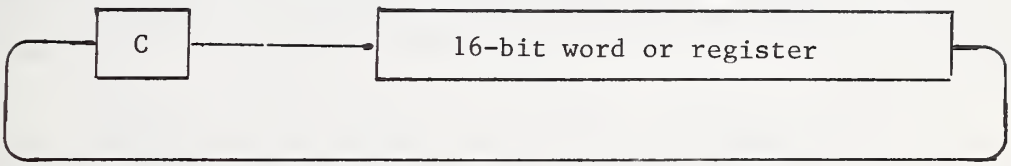


Figure 5. Shift/Rotate Instructions Work On 17-bits.

Some minis may incorporate a post-shifter that operates on the result after it has passed through the arithmetic/logic unit. This gives the instruction set added power at little extra cost. Programming complexity and unreadability are increased, however, in exchange.

A basic repertoire can be very basic, indeed. Remember that mini hardware is modular and additional modules are added to increase performance for a particular application. For example, minis with a central bus organization may replace software floating point routines with a floating point module that attaches to the bus. Automatic address translation and memory protection could be added to the basic emulator of a micro-programmed mini by upgrading the control memory.

Minicomputers placed in harsh environments would require an optional power fail/auto restart feature that saves important restart information when power failure is detected. Other options include exception handling for overflow, memory parity, and hardware failure detection, and a real-time clock or interval timer for event-driven applications.

A large number of I/O driven systems are implemented on minicomputers. For this reason a final word on minicomputer I/O subsystems is given. A minicomputer programmer must deal with physical level I/O more often than a systems programmer working on a large system. Therefore he needs to know the variety of options available to him.

The three most important parameters in I/O subsystems are (1) maximum transfer rate, (2) interrupt structure, and (3) type of I/O: program controlled or direct memory access, DMA. Transfer rate is determined by the designer. The minicomputer programmer must accept whatever rate is available. The interrupt structure is also built-in, but a programmer may choose to use it or circumvent it.

Program controlled I/O is used for console devices or slow peripherals that transfer limited volume of data. The mini is synchronized with the device through an elementary handshake that uses two registers: STATUS and DATA. A typical input is made as follows:

(1) Start Read operation

(2) Test STATUS and wait until device is ready (loop)

(3) Transfer data from DATA to main memory or a working register

The loop in step (2) holds up the entire computer until a transfer is made. This same I/O operation can be performed concurrently with the operation of the cpu if an interrupt is enabled in step (1). It is the responsibility of the programmer to provide a program to service the interrupt when it occurs some time later.

DMA (direct memory access) transfer is accomplished by stealing memory cycles from the cpu, that is, by accessing memory in between

the times when the cpu is accessing memory. In the event of a conflict, the DMA device overrides the cpu, locking it out, temporarily. (The cpu can wait, but a device often cannot.) Although DMA transfer rates are potentially very high, the cpu may be slowed or stopped during such fast transfers.

DMA I/O may be initiated and completed just as the program controlled I/O, but this is wasteful of resources. Instead, it should be carried out concurrently with the execution of the program and synchronized by an interrupt or a flag (register). The steps are:

(1) Program initializes an ADDRESS register and COUNT register within the DMA unit (an additional hardware module).

(2) Program signals device to START.

(3) The DMA unit keeps track of the transfer by decrementing COUNT and testing it for zero.

(4) DATA is transferred to/from memory location specified by ADDRESS which is incremented by the DMA unit.

Once the DMA transfer is completed either an "I/O completed" interrupt occurs (if the programmer enabled interrupts) or else the STATUS register changes the value to DONE. In either case, this method is well suited for block transfers at high speed.

We can gain additional processing speed if interrupt driven (triggered) scheduling is used in a program. This allows simultaneous use of devices with the minicomputer cpu.

A variety of schemes exist for interrupt servicing, but in this discussion we define an interrupt as a hardware-forced call to a subroutine. The subroutine is called a service routine because it services the interrupt.

The service routine performs a service (input/output, buffer switching, or transforms tables), restores the processor status, dismisses the interrupt by clearing flag bits, and returns control to the interrupted code.

In a simple interrupt structure the minicomputer system reserves main memory locations 0000, 0002, and 0004 for the address of the service routine, the "old" processor status word, and the return address, respectively. Then, when an interrupt occurs a branch to the location specified in 0000 is executed and at the same time the status word and program counter are saved on 0002 and 0004.

Upon return from the service routine, the status word and program counter are restored from the save areas in main memory. Obviously, this system breaks down if subsequent interrupts occur while the cpu is

executing in a service routine, unless the interrupt is held (at a level) until the current one is dismissed (the saved status word and program counter are destroyed when the second interrupt occurs). Another disadvantage is that the service routine must do work just to determine which device caused the interrupt, since all devices pass control through reserved location 0000.

In more sophisticated minis a collection of dedicated words of main memory are used instead of 0000, 0002, and 0004. Each triple performs the same function as described for the simple I/O system, but instead there is a unique triple for each device. These dedicated words are often called I/O VECTORS; see Figure 6.

A separate vector and separate routine are provided for each device. There is no possibility of lost information if an interrupt occurs during the processing of an interrupt, since subsequent interrupts save the old program status and counter in the I/O vector. The only exception to this prevention occurs if there is a subsequent interrupt caused by the same device that is currently being serviced through the I/O vector.

On minis that use a pushdown stack, the PS and PC information is saved (pushed) on the stack instead of in the vector. In this situation repeated interrupts on the same vector are allowed because duplicates of PS and PC are saved on the stack. Thus, a combination of I/O vector and pushdown stack seems to offer the greatest power for minicomputer interrupt structures.

Another scheme not yet mentioned is the Interrupt Mask Register system. This uses a single register containing a zero in bit position p if no interrupts are allowed by device #p, or else a one if interrupts are allowed. The program or hardware must check each bit in the Mask Register to determine which device will be serviced.

2.5 Summary

The architectural trends for minicomputers are microprogramming*, central bus structure, variety of register organizations, sophisticated I/O structures, and special purpose instruction sets. These are outgrowths of the current minicomputer attitude. This attitude is characterized by a typical minicomputer with:

---

*For another view, see R. F. Rosin, "The Significance of Microprogramming," Proc., International Computer Symposium, 1973, A. Gunther, et al (editors), North-Holland Pub. Co. (Amer. Elsevier for U.S.A. - N.Y., N.Y.).

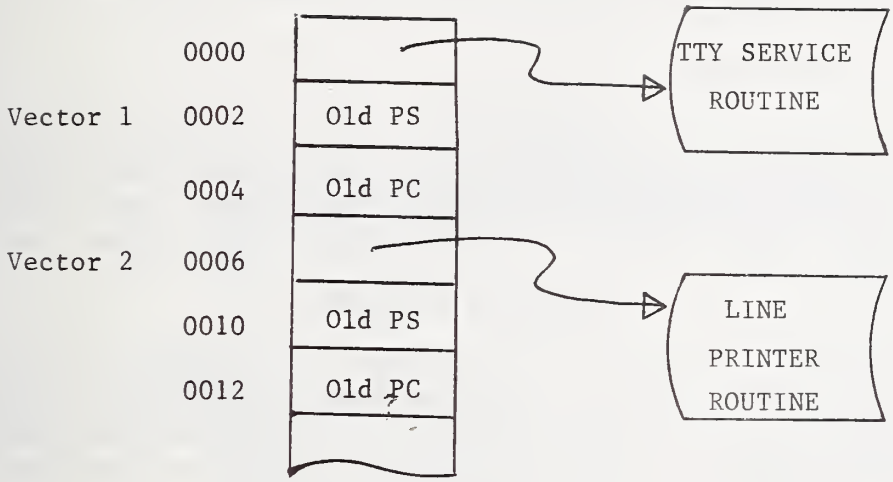


Figure 6. I/O Vectors For Automatic I/O. They Contain Pointers To Service Routines And Keep The Old PS And PC Values.

(1) General purpose register, bus structure

(2) 16-bit, two's complement integer arithmetic

(3) Data manipulation (byte and word) instructions that are limited in shift/rotate, test/branch, and logical operations.

(4) Small memory, proliferation of addressing modes and protection options.

(5) Variety of add-on options

The outgrowth of this attitude will be to overcome limitations as stated above. Again we reiterate that the sum total of these limitations is the difficulty with which minis are programmed. The challenge in the next ten years will be to minimize software development costs.

3. MINICOMPUTER PROGRAMMING

In the previous sections of this report, we saw that minicomputing is an attitude about smallness. Smallness implies limited programming, and yet it is possible to develop programs of respectable size that run on minicomputer hardware. To do so, however, requires software tools specifically adapted to the minicomputer attitude.

The search for powerful software tools is further clouded by a variety of minicomputer organizations embodied in hardware. We emphasized the lack of a measure of architectural effectiveness in the last section, and in this section we can only point out intuitive or aesthetic measures of software effectiveness. For this purpose we examine three machines. Mini A is a special purpose register, single-address machine (a simple mini) [8]. Mini B is a general-purpose-register, two address machine (typical) [5], and Mini C is a stack machine (microprogrammed) to support a high level systems implementation language [4].

3.1 Case Study I: Mini A (simple machine)

Mini A is a commercially available computer that is designed to be low cost, fast, and uncomplicated. It is organized around a 17-bit distributed bus as shown in Figure 7. The bits are cycled through the ALU (arithmetic/logic unit). The result out of the ALU is (possibly) shifted by one bit position. A skip sensor determines where the next instruction is fetched. The Load/Noload switch determines whether the 17-bit bus data is placed back into a register or lost. The C-bit is a carry, overflow bit that is input into the ALU along with the 16-bit register values.

The registers are dedicated to accumulator or index functions in addition to their special purpose assignments given below.

AC0 16-bit, one - or two's - complement accumulators

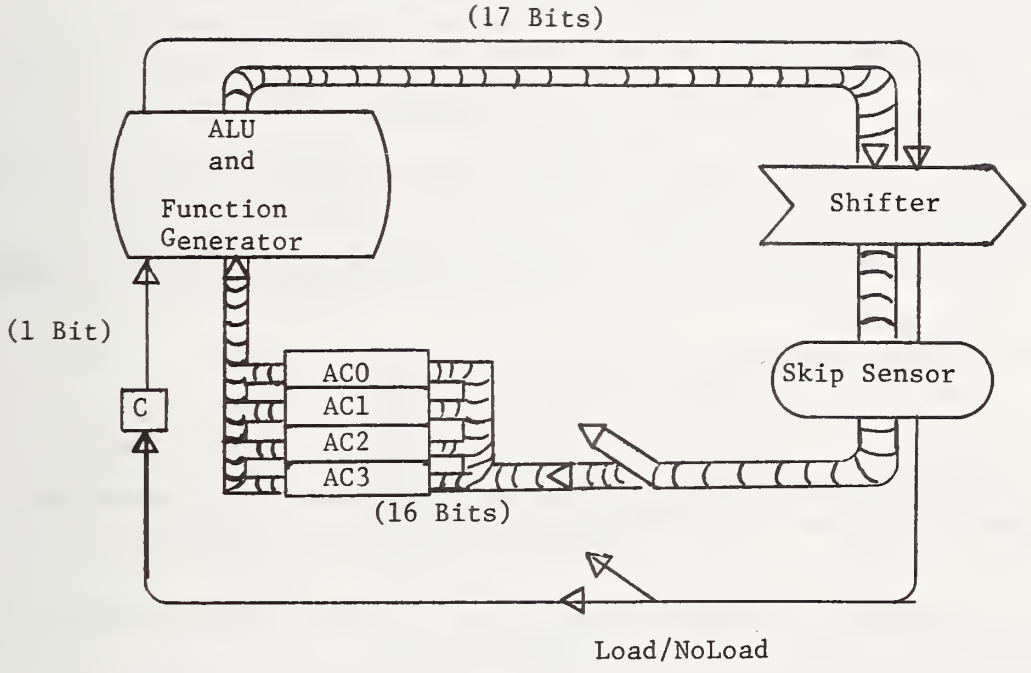


Figure 7. Mini A. A simple Minicomputer with 4 Special Purpose Registers, Shifter, Skip Sensor, ALU, and Bus.

AC1 used in arithmetic/logical operations

AC2 16-bit, one - or two's - complement
AC3 accumulators exactly like AC0 and AC1, except these two registers are also index registers.

AC3 A second purpose for AC3 is as a subroutine return address register. The return address is stored here.

In addition to these special purpose registers there are special purpose words in main memory that are used in autoindexing. They assist in indexing through arrays by automatically increasing or decreasing in value each time they are treated as a pointer to data.

LOCATIONS: 020_8 to 027_8 Autoincrement
030_8 to 037_8 Autodecrement

The hardware determines that these locations are being used as a pointer whenever an indirect address is computed through these locations.

The instruction set of Mini A is partitioned into four classes:

Move Data
Memory Reference
ALU
I/O

We will demonstrate programming techniques using assembler mnemonics and avoid the necessity of defining the machine codes for each operation.

Operands are accessed through one of five addressing modes as follows (PC is the program counter):

Page Zero: Direct access to locations 000_8 to 377_8
PC-relative: Contents of PC plus displacement
Indexed: Contents of ACi;i=2,3 plus displacement
Indirect: Address of address when bit zero is set
Autoindex: Use when indirect addressing location 020_8 to 037_8

The shifter and carry bit is controlled through additional bits in each instruction word. We indicate CARRY/SHIFT control with an assembler mnemonic as follows:

Z: clear CARRY register (zero)
O: set CARRY register (one)
C: Complement CARRY register (reverse)
L: rotate the 17-bit bus left one bit
R: rotate the 17-bit bus right one bit
S: swap low order 8-bits with high order 8-bits

These mnemonic codes are appended to the right-hand end of any op-code mnemonic. They are suffixes to the basic instruction mnemonics.

The skip sensor is also controlled by placing skip codes in the operand field of assembler statements. A few of the codes are given below:

SZR: skip next instruction if zero result.

SZC: skip next instruction if carry is zero.

SKP: skip next instruction

Combining the codes for CARRY/SHIFT/SKIP control with the functions of the ALU results in powerful instructions for this simple machine. The abundance of suffixes may damage readability, though, and standard programming techniques for Mini A border on programming trickery. Let us consider a few examples.

The MOV instruction copies the contents of one register into another (or same) register. When used in combination with other functions it is extremely versatile.

```
MOV  1,1,SZR;      test AC1 for a zero result and skip the next
                   instruction if AC1 contains a zero.

MOVL# 1,1,SZC;     # means Noload.  A copy of AC1 is shifted left
                   one bit (rotation of 17 bits) and the next
                   instruction skipped if C=0.  This tests the
                   sign bit of AC1.

MOVOR 1,1,SKP;     Put a 1 in C, then rotate AC1 right.  Skip the
                   next instruction.

MOVZR 1,1;         Zero the C bit, then shift right.
```

3.2 Example A.1

Suppose we wish to divide a two's-complement number by two. There are two cases: when the number is positive and when the number is negative.

```
MOVL# 1,1,SZC;     test the sign bit and skip if it is zero

MOVOR 1,1 SKP;     it is negative, so right shift (divide by 2) and
                   put in a 1 (sign bit), and then skip

MOVZR 1,1;         it is positive, so right shift and put in a zero.
```

This example shows how test and branch instructions are incorporated into a single copy instruction. Now study the following example

that shows how the autoindex locations in main memory function. The assembler format is:

: denotes a label

; denotes a comment

, denotes an operand

The assembler mnemonics for the next example are:

LDA	load a register from memory
STA	store a register into memory
DSZ	decrement and skip if result is zero
JMP	jump

3.3 Example A.2

The following subroutine copies a block of data (30 words) from locations 2000_8 in reverse order to locations 5205_8 to 5150_8. The autoincrement and autodecrement words are used with an indirect bit (@ in assembler) to indicate that they are pointers.

```
      MOVE:  LDA     0,CNT      ;Set-up autoincrement...
             STA     0,21       ;... in location 21(8).
             LDA     0,CNT+1    ;Set-up autodecrement...
             STA     0,35       ;... in location 35(8).
      LOOP:  LDA     0,@21      ;Get a word and...
             STA     0,@35      ;... move it.
             DSZ     CNT+2      ;decrement and test...
             JMP     LOOP       ;... otherwise, loop again.
             JMP     0,3        ;return to main through AC3 (indirect
                                 jump)
      CNT:   001777             ;2000-1=pointer to...
                                ;... table to be moved
005206                          ;5205+1=pointer to...
                                ;... destination.
000036                          ;length of data (30).
```

The example above gives the reader an idea of how an assembler program appears in this simple machine. The LDA and STA mnemonics cause AC0 to be loaded and stored. The @ bit causes location 21_8 to be auto-incremented during data fetching, and location 35_8 to be autodecremented during the move.

The I/O instructions of Mini A are part of the skip, no-op, and data transfer instructions. To demonstrate this, in the next example a byte of data is read from the system console.

3.4 Example A.3

A single byte of data may be read into AC0 with the following code. Remember the three steps of program controlled I/O discussed earlier. TTI is the device number of the console.

```
START:  NIOS    TTI     ;start the terminal, READ.
        SKPDN   TTI     ;skip next instruction...
                        ;...if TTI is ready.
        JMP     .-1     ;loop back (not ready).
        DIAS    0,TTI   ;copy into AC0.
```

The codes may be interpreted by the reader as follows. NIOS is a no-op, NIO, plus a suffix: S=set BUSY flag in the device STATUS register. The SKPDN code means to "skip next instruction if the DONE bit is set" in the STATUS register. The DIAS instruction actually performs the transfer from device TTI to AC0.

An experienced Mini A programmer would acquire a bag of techniques most likely called tricks by other programmers. This makes the task of software development and documentation an ordeal at best. It is useful to be aware of these techniques simply to assist anyone wishing to understand Mini A software. Consider the following special techniques.

3.5 Special Techniques

```
SUB0    AC,AC     ;clear AC and CARRY
SUBC    AC,AC     ;clear AC but save CARRY
SUBZL   AC,AC     ;generates a +1 in AC.
ADC     AC,AC     ;generates a -1 in AC.
ADCZL   AC,AC     ;generates a -2 in AC.
NEG     AC,AC     ;this pair of codes...
COM     AC,AC     ;...decrements AC by one.
```

To help the reader see why the above sequences of code work, let us examine the ADCZL code. The ADC part causes the one's complement of AC to be added to the contents of AC, itself. The Z suffix causes the CARRY bit to be cleared. The L suffix shifts the 17-bit result left so that the low order bit (bit 15) is a zero and the carry out from the high order bit is placed in C. Adding the one's - complement of a number to itself creates a two's - complement (-1). Shifting in a low order zero produces two times negative one, i.e. (-2).

3.6 Case Study II: Mini B

Mini B represents a compromise between sophistication and low cost. One of the design goals of Mini B was to make assembly language programming easy. The machine is organized around a universal bus capable of transmitting 18-bit address words and 16-bit data words.

Figure 8 demonstrates how each system unit is modularized and connected to other units through the universal bus.

Mini B hardware is more complicated than Mini A, but as we will see its programming complexity is less than that of Mini A because of the straightforwardness of the instruction set. The nine registers shown in Figure 8 are the key to understanding Mini B.

```
R0 through R5          :general purpose registers used for
                        arithmetic/logic, indexing, and as
                        pointers to data.

R6 = SP                :used by the hardware as a pointer
                        to the top element of a stack
                        stored in main memory.

R7 = PC                :the program counter.

PS                     :the processor status word.
```

In addition, the PS register contains testable flags (condition codes) for various conditions. For example, if a result of a computation or test (TST) is zero, Z = 1 (See Figure 8). The T = TRAP bit causes an interrupt after the execution of each instruction. This is used for debugging purposes.

External interrupts are allowed only by devices of higher priority than the cpu (see discussion of bus priority). Therefore, the PRIORITY field of the PS register determines the level at which the cpu currently runs. For example, if device #1 on the bus wishes to transfer data to main memory, the bus is relinquished only if the cpu is in a low enough PRIORITY state.

Mini B is a two-address 16 bit word machine. This means that each instruction operates on a SOURCE and DESTINATION operand. The format of an assembler instruction is:

```
optional label  :  op-code mnemonic    SOURCE, DESTINATION; Comment
```

This format is shortened to a single operand when the SOURCE equals the DESTINATION. Examples of single operand instructions are given below.

```
CLR     DESTINATION     ;clear DESTINATION
COM     D               ;one's-complement D.
INC     D               ;add one to D.
DEC     D               ;subtract one from D.
NEG     D               ;two's-complement D.
TST     D               ;test D, set condition code.
ROR     D               ;rotate D right one bit position.
```

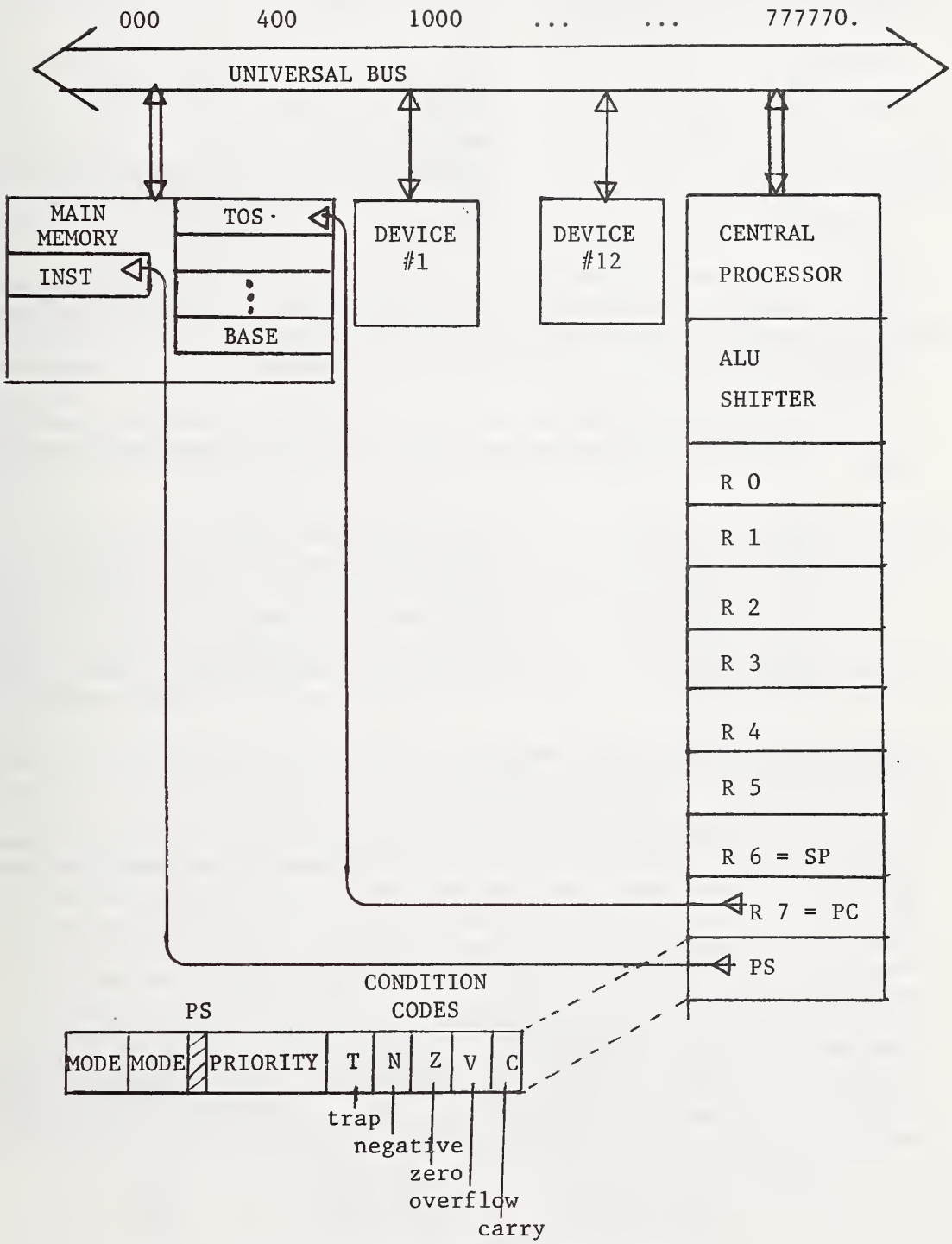


Figure 8. Mini B As The Programmer Sees It.

```
ROL     D               ;rotate D left one bit position.
ASR     D               ;divide D by two.
ASL     D               ;multiply D by two.
SWAB    D               ;exchange high and low order bytes.
ADC     D               ;add C to D (carry add-in).
SBC     D               ;subtract carry from D.
```

Each of these instructions operates on a full word (16 bits) of data. They may be converted into byte operations that manipulate half-words merely by suffixing a B to each instruction mnemonic. Thus, CLRB causes a single byte at DESTINATION to be cleared.

The double operand instructions are truly two-address instructions. In the following, a byte-operand option is specified with brackets. SS means source and DD means destination, for brevity.

```
MOV[B]  SS,DD           ;copy from SS to DD.
CMP[B]  SS,DD           ;compare SS minus DD.
ADD     SS,DD           ;add:  DD←DD+SS.
SUB     SS,DD           ;subtract:  DD←DD-SS.
BIT[B]  SS,DD           ;bit test:  SS   .AND.   DD.
BIC[B]  SS,DD           ;clear bits masked by SS:  DD←¬SS.AND.DD.
BIS[B]  SS,DD           ;set bits masked by SS:  DD←SS.OR.DD.
```

In addition, Mini B has a variety of branch instructions whose meanings will be obvious in later examples.

Each instruction stored in memory is either one, two, or three words long. The length of instruction depends on the addressing mode employed. If the SOURCE and DESTINATION are both registers, then a single word instruction is generated from the assembler mnemonic. If only one of the operands is a register, then two words are generated. When both operands are main memory references, then three words of instruction information are generated. In the assembler notation, a per cent sign, %, distinguishes a register from a memory location. Note also that the equal sign is used to perform equivalence in the assembler. Before continuing to examples, the reader should study the variety of addressing modes made possible by treating PC as an index register simultaneously with its dedicated function as a program counter.

MODE (Name)	SYMBOLIC	DESCRIPTION
register	R	Contents of R is the operand
R deferred	(R)	Contents of R is a pointer to operand
autoincrement	(R)+	R is pointer, incremented after use
autoincrement, deferred	@(R)+	Contents of R is the address of the address
autodecrement	-(R)	Decrement R, use as a pointer
autodecrement, deferred	@-(R)	Decrement address of pointer
index	X(R)	Contents of R plus X is address
index deferred	@X(R)	Contents of R plus X is address of address
immediate	#n	n follows instruction as data
absolute	@#n	n is an absolute address
relative	n	relative displacement to n follows
relative deferred	@n	address of address

Perhaps the most interesting feature of Mini B is its incorporation (perhaps half-heartedly) of a pushdown stack. Actually any one of the working registers can be used as a stack pointer. Register six, however, is used by the hardware as a stack pointer during calls to subroutines and during external interrupts. The programmer may also use R6 = SP for storage of temporary results or for passing parameters to a subroutine.

The stack top is limited to location 400_8 in main memory but its base may be placed anywhere the programmer desires. Suppose we wish to set the base at location 477_8 and allow the stack to grow toward location 400_8. We could do this with the MOV instruction using the immediate addressing mode.

```
MOV         #477,%6         ;set stack base.
```

A push or pop operation is performed by MOVing data to/from a register using the autoincrement or autodecrement addressing modes. Observe that the stack increases in length by decreasing the address stored in register six.

```
MOV         %0,-(%6)        ;push R0 onto stack.
MOV         (%6)+,%0        ;pop TOS (top-of-stack) to R0.
MOV         (%5),(%6)+      ;delete TOS.
```

The stack is also used to hold the return address linkage during a subroutine call. The JSR instruction always pushes the contents of some register on the stack. If that register is the PC then the return address is pushed.

```
      PC = %7

      JSR          PC,SUB      ;call SUB and place.
                               ;PC on stack.
      RTS          PC          ;pop TOS into PC.
```

Notice the use of the "=" assembler pseudo-operation to improve readability of the code (PC = %7).

In the previous section we promised to simplify I/O with a universal bus structure. In fact there are no special I/O instructions in Mini B as shown below. Let PRS be the universal bus address of a STATUS register. Let PRB be the bus address of a DATA register. In this example, #100200 is a bit mask for testing the error condition (#100000) and ready condition (#000200).

```
      INC          PRS         ;start READ operation by setting
                                bit zero.
WAIT: BIT          PRS,#100200 ;test bits 15 and 7.
      BEQ          WAIT        ;branch if equal to zero.
      BMI          ERROR       ;branch if minus high order bit is
                                sign.
      MOVB         PRB,(R5)+   ;copy from DATA register . . .
                               ;. . . to buffer area.
```

The preceding segment of code demonstrates the use of the INCrement instruction to start a read operation in some device. The device is chosen by setting PRS and PRB to some (predetermined) bus address. The BIT instruction ANDs the mask with the status register and sets a condition code only; the PRS is not affected. The next two branch instructions act on the condition code set by the BIT instruction. Finally, when the device is error-free and ready, a single byte is MOVed from PRB to the buffer location pointed at by register five. R5 is incremented by one (unless R5 = %6 because the SP register is always autoincremented and autodecremented by two to maintain word alignment).

3.7 Example B.1

Suppose we program Mini B to perform a communications function. In such an environment data is being read into the mini, manipulated, and output. In a communications application it may be necessary to develop high speed algorithms (the minicomputing attitude) to

perform bit reversal as in a fast Fourier Transform application[+] or simply to generate a parity bit for error control. For example, counting the number of set bits in a word is a necessary part of computing a parity bit. How can we perform this simple counting operation at high speed? Consider the following algorithm:

1. Let $W = w_0\ w_1\ w_2 \ldots w_n$ be the bit string of cleared or set bits. We want to compute $\Sigma\ w_i$ by counting. Set KOUNT = 0.

2. Repeat the following no more than n times:

   2a. Test W to see if it is zero. If W = 0 the algorithm is done and KOUNT contains the number of set bits. If $W \neq 0$ go on to the next step.

   2b. Compute W ← (W).AND.(W-1)*

   2c. Increment KOUNT.

3. Stop.

This algorithm works best for values of W such that there are few (less than one-half) set bits. The Mini B program follows:

```
KOUNT = %0                          ;assign variables.
W     = %1                          ;W is passed thru R1.
TEMP  = %2                          ;temporary scratch req.
BITNO:                              ;entry to subroutine.
              CLR       KOUNT       ;set KOUNT = 0.
LOOP:         TST       W           ;done?  test value of count
              BEQ       DONE        ;yes (i.e. branch if code z = 1)
              MOV       W,TEMP      ;no.  compute W-1 . . .
              DEC       TEMP        ;. . . and save in TEMP.
              COM       TEMP        ;fake an AND . . .
              BIC       TEMP,W      ;. . . with COM, BIC.
              INC       KOUNT       ;KOUNT = KOUNT + 1
              BR        LOOP        ;Repeat
              RTS       %7          ;return.
```

[+]The fast Fourier Transform algorithm employs a clever index algorithm that is implemented in a manner similar to the example.

*Let $W = 1\cdot2^a + 1\cdot2^b + \ldots + 1\cdot2^{l'} + 1\cdot2^l$ where $a>b>\ldots>l'>l\geq0$
then $W-1 = 1\cdot2^a + 1\cdot2^b + \ldots + 1\cdot2^{l'} + 0\cdot2^l + 1\cdot2^{l-1} + 1\cdot2^{l-2} + \ldots + 1$ or clearly then W.AND.W-1→W
gives $W = 1\cdot2^a + 1\cdot2^b + \ldots + 1\cdot2^{l'}$ as required. W = 0 when KOUNT is correct.

Notice in the program above that COM and BIC, taken together, perform a logical AND operation on TEMP and W. This is required on Mini B because the BIT instruction does not store a result. Instead, to get a result from an AND operation two instructions are substituted in place of the BIT instruction normally used for ANDing.

3.8 Special Techniques

Mini B programmers need not acquire a special techniques repertoire as needed for Mini A. The limited instruction set may cause distress, however. For example, the smaller models often lack an exclusive OR, multiply, divide, and logical AND operation that produces a result. The programmer must develop a collection of macros to simulate these instructions.

The consistency and format of assembly statements makes programming Mini B straightforward. The machine is designed with this in mind. In most cases, the operands may be register or memory locations and the programmer need not worry about the actual resource being used. This eases the programmer's burden, but note that program length is increased when two and three word instructions are used in place of register-to-register instructions.

The added power of recursive subprograms and stack processing probably reduces program size in many applications. This is demonstrated by the short program segment below that strips off decimal digits, one-at-a-time, in preparation for output to a terminal. The segment is recursive.

3.9 Example B.2

```
CONVRT:    JSR     PC,DIVIDE      ;divide (QUOTIENT/10).
           MOV     REM,-(SP)      ;save REMainder on stack.
           TST     QUOTIENT       ;test quotient.  Done?
           BEQ     DONE           ;Yes.
           JSR     PC,CONVRT      ;No.  call self (recursively).
DONE:                             ;Output the REMs . . .
                                  ;. . . stored on the stack.
```

This example shows how a compact conversion routine is programmed using the full power of stack processing on Mini B. As an example, assume QUOTIENT = 123, initially. After division by 10, the QUOTIENT becomes 12 and REM = 3. Repeated recursions produce 2, 1, and then zero. The binary equivalents of 3, 2, 1 are stored on the stack in reverse order. When they are popped, the binary coded digits are ready for output in the proper order.

3.10 Case Study III: Mini C

Minicomputer A is low cost and simple. System complexity is dealt with by the software and not the hardware. Minicomputer B is more

sophisticated and, by nature of its organization absorbs more complexity at the hardware level than Mini A. Even so, both require programming talent of a greater caliber than is expected of programmers on many large machines. Therefore Mini A and Mini B do not represent great advances in reducing programming effort.

Mini C is presented in this report because it represents one approach to reducing programming costs in minicomputer systems. Clearly, the relative cost of a minicomputer cpu is negligible when compared to the cost of peripherals and software. The cost of peripherals is expected to drop in the same way that mini cpu costs have declined. The major problem faced by minicomputers in the next ten years is programming and software complexity.

The basic philosophy of Mini C is simply this: reduce programming costs by forcing complexity into the cpu hardware (actually firmware). When this is done the programmer no longer develops software in assembly language. Instead, a high level machine oriented language is used to develop systems software, language translators, text editors, application packages and utilities.

Mini C is a minicomputer specifically designed to efficiently support a high level systems implementation language [4,7]. The language is a PL/1 derivative, that is, it is block structured, procedure oriented, and equipped with control constructs for modular (structured) programming.

Mini C emulates an execution environment that supports the most flexible type of process, i.e. the pure process. This is accomplished by protecting a running program from WRITE infringement (the program is temporarily in a ROM space) and guaranteeing data independence for every active process. Figure 9 shows how a program segment and a data segment are delimited in main memory.

In Figure 9 the currently executing program of length PL resides in main memory locations PB to PB + PL. The next instruction will be taken from location PB + PP. This space is designated ROM during the program's execution.

The data space for the program shown in Figure 9 is a push-down stack located at SB with length SL. All operands are taken from this stack (with two exceptions) during program execution. SB + EP points to a block of 4 words on the stack called a MARK. MARK contains all the pertinent information required to establish a LOCAL ENVIRONMENT for the corresponding block currently executing in program space. Each time a new block (or procedure) is entered in the program, a MARK is pushed onto the stack. Each time a block is exited in the program, the stack is rolled-back (the MARK is popped). The MARK is intimately related to the high level language discussed later.

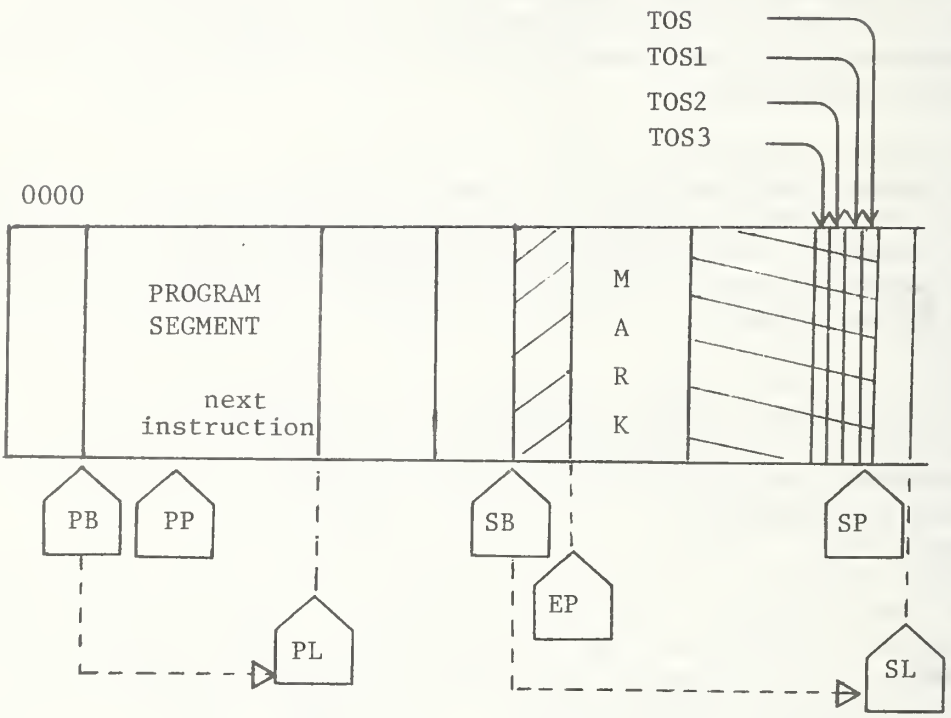
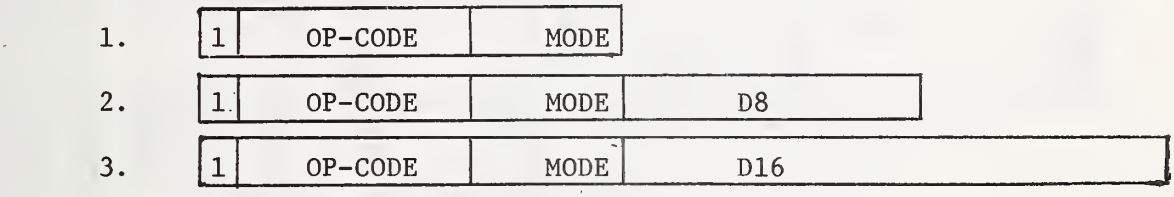


Figure 9. The Program Space And Data Space For Mini C Main Memory.

Emulated Registers:

PB ≡ prog base
PL prog length
PP pts to current inst
SB stack base
SL stack length
EP envir ptr
SP stack ptr to top of memory stack

The machine instructions generated by translating the high level language (HLL) will generally be either one, two, or three 8-bit bytes depending upon the addressing mode. The single byte instruction acts on the TOS (top-of-stack) elements, only. The other two instruction formats include fields containing displacement information.



There are eight addressing modes as shown in Figure 10. A running program would be executing within the environment established by MARK (EP is the environment pointer). All data beyond the MARK is called local data. Modes 2 and 3 provide access to these local values by computing the sum of SB + EP + D8 and possibly the TOS value.

Global values are values outside the present environment but within the current program's data stack. For example, mode 5 computes an address by summing the SB + (TOS) + (TOS1) values (TOS1 is the element next to TOS in the stack). Top-of-stack comprises 5 (emulated) registers

TOS
TOS1
...
...
TOS4

in descending order.

Absolute addresses may be accessed only by privileged instructions. Assuming that a given program is an executive routine of some sort, absolute addresses are computed from TOS plus four times the next element on the stack, TOS1.

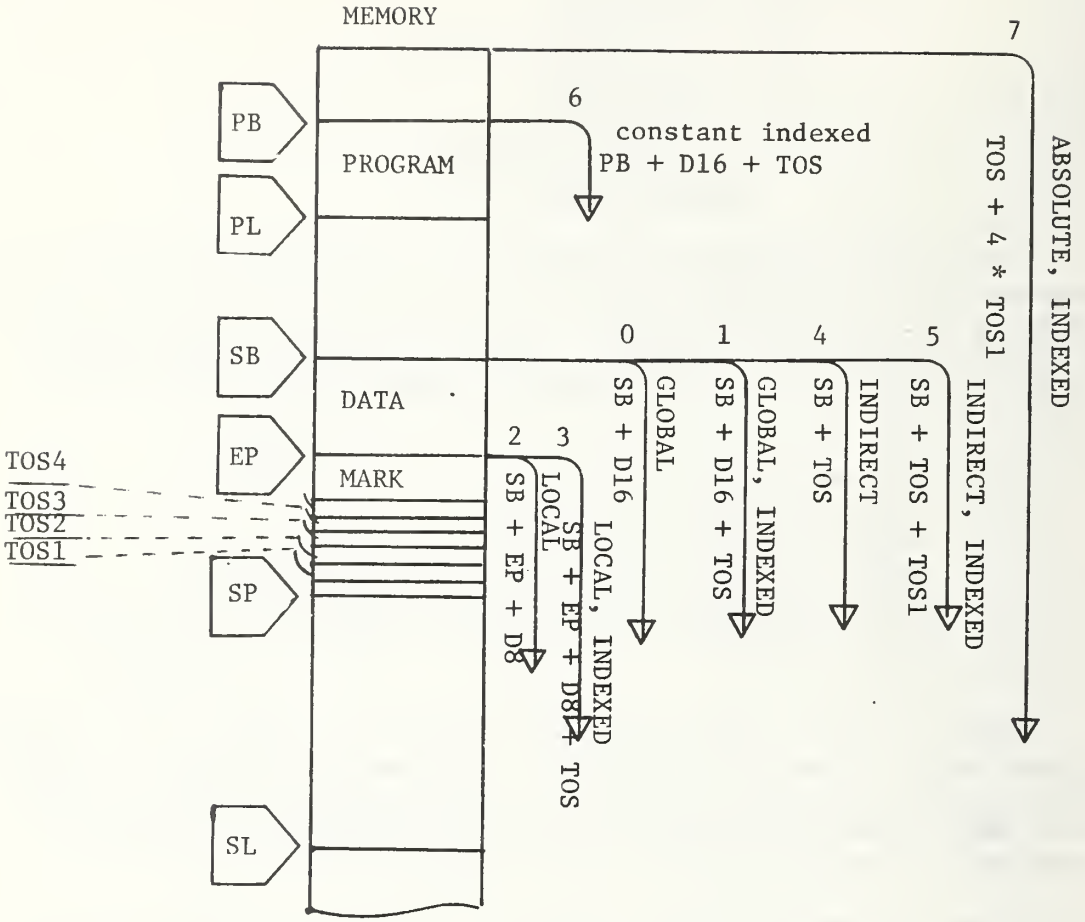


Figure 10. The Addressing Modes of Mini C.

Program constants can be read from the program space using mode 6. The protection mechanism for program segments prevents a write operation (pop) from occurring in mode 6.

Now let us digress for a moment to establish a motivation for the complexity presented above. Mini C is designated to reduce programming effort through the principle of forcing complexity into the hardware. Actually, Mini C is microprogrammed to operate on the stack space and program space in the way presented above. Therefore, complexity is forced down to the firmware level of the machine. We will reap the benefits of this principle by never having to worry about firmware complexity while programming. This is accomplished through the HLL mentioned earlier.

Figure 11 is a listing of a program written for Mini C. It is a simple example of a main program and an internal procedure. The application is purposely simple because we are more concerned with understanding the relationship between this program and the hardware described by Figures 9 and 10. Procedure SUM computes the sum of elements in ARRAY and is not shown.

The rules of this language are obvious to any PL/1 programmer and will not be belabored here. Simply stated, they are as follows:

1. Unlike PL/I, all variables are typeless.
2. All variables must be declared.
3. The scope of all variables is the block in which they are declared.
4. Simple parameters are passed to procedures by value, arrays are passed by address.
5. Variables are declared as 16-bit WORD, 8-bit BYTE, POINTER, or PROCEDURE valued.
6. Arrays index from zero to their upper bound.

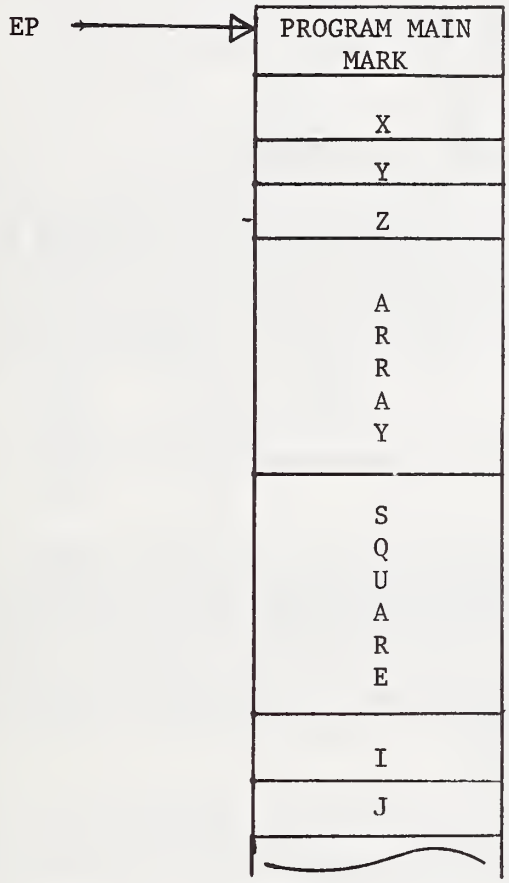
Let us study step-by-step how the program of Figure 11 is executed by the Mini C firmware. An assembler-like mnemonic code is used in place of machine code to make understanding easier. Each mnemonic is explained as required.

The PROCEDURE statement produces a MARK on the stack as shown above. The DECLARE statement reserves storage space on the stack. Note that EP points to the MARK that defines the current environment. At this point the mini must set SP and begin executing the DO - loop.

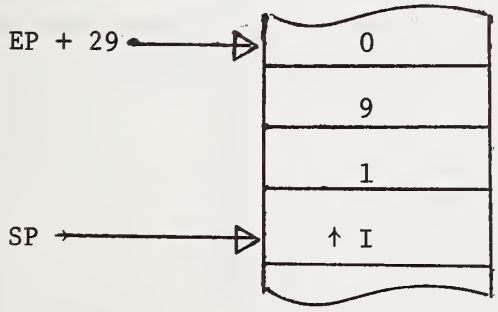
```
PROCEDURE MAIN;
DECLARE SUM   EXTERNAL PROC WORD,
        (X,Y,Z)             WORD,
        ARRAY(9)            WORD,
        SQUARE(9)           WORD,
        (I,J)               WORD;

    DO I=0 TO 9;
    ARRAY (I) = I;
    END;

X = SUM(ARRAY);

    DO J=0 TO 9;
    SQUARE(J) = ARRAY(J) * ARRAY(J);
    END;

Y = SUM(SQUARE);
Z = Y/10 + X/10;
END;
```

Figure 11. A Sample HLL Program For Mini C.

```
DECLARE      SUM              EXTERNAL PROC WORD,
             (X,Y,Z)                        WORD,
             ARRAY(9)                       WORD,
             SQUARE(9)                      WORD,
             (I,J)                          WORD,
```




```
DO    I = 0 TO 9;
      SSP   29
      L0
      L9
      L1
      LADR  0,0,0,1,I
      DIB   24
```



set stack pointer register to EP + 29

load 0, 9, and 1

load a pointer to I

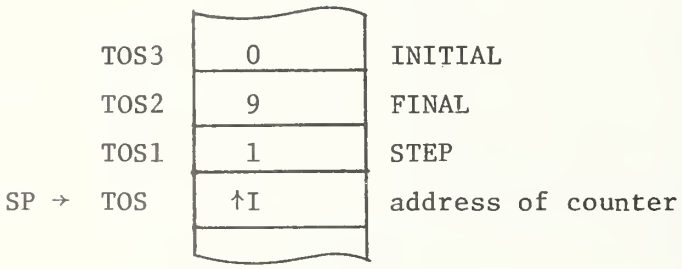
The DO-loop is set-up by pushing the INITIAL value, TEST value, INCREMENT value, and address of loop counter I onto the stack. This is shown above by the SSP (set stack pointer) and L (load) instruction mnemonics corresponding to the DO-loop statement in HLL.

```
ARRAY(I) = I;

        LW      0,I
        LW      0,I
       STW      1,ARRAY
```

The LADR instruction causes the address of variable I to be pushed onto the stack. The DIB mnemonic indicates that the variable at the location specified by TOS will be tested as follows: TOS1 (next word under TOS in the stack) is treated as the INCREMENT value for the loop counter. TOS2 is the test value and TOS3 is the INITIAL value.

The DIB (DO-loop initialize and branch) instruction performs the following steps:

DIB 1. Copy the INITIAL value into location I and,
2. If INITIAL > FINAL then pop TOS, TOS1, TOS2, TOS3, and branch to PB + 24.



The two LW (load word) instructions push first the right-hand-side of the assignment statement and then the subscript of ARRAY onto the stack. Finally, the assignment is carried out by the STW (store word indexed) instruction. The stack is cut back to the address of I.

The loop is tested with the DSBB instruction (Do-loop step and branch back). This instruction is generated by the END.

```
END;

        DSBB        10
```

DSBB:
1. Step through TOS; hence I=I+1. I=I+ TOS1≡(I)+1
2. Compare I and FINAL, that is compare the value accessed by addressing through TOS with the value stored at TOS2.
3. If I is less or equal than FINAL then branch back 10 bytes.
4. If I is greater than FINAL then pop TOS, TOS1, TOS2, and TOS3.

The next segment of program executes a procedure call. This call will force a new environment onto the machine by establishing a stack MARK. The parameter ARRAY is passed as an address in order to conserve memory in the local environment of SUM.

```
X = SUM (ARRAY);

    MARK     0,1,1, SUM
    LADR     0,0,0,1, ARRAY
    CALL     4, SUM
    STW      0,X
```

The MARK establishes a new environment and provides backward pointers so that the stack can be rolled-back upon return from SUM. Immediately following the four word MARK is the address of ARRAY. A mode 4 CALL is executed which passes the address of ARRAY to a new program segment called SUM. Upon return from SUM, the new value of X is on the stack, and this value is stored (STW).

By now the reader will have some idea of the basics of Mini C. Therefore, the remainder of the translated program is presented below. The correspondence between HLL and the segments of machine code are obvious.

```
DO      J = 0 TO 9;
        L0
        L9
        L1
        LADR      0,0,0,1,J
        DIB       68

SQUARE (J) = ARRAY (J) * ARRAY (J);
        LW        0,J
        LW        0,J
        LW        1, ARRAY
        LW        0,J
        LW        1, ARRAY
        MUL
        STW       1, SQUARE

END;
```

```
        DSBB    20
Y =  SUM (SQUARE);
        MARK    0,1,1,SUM
        LADR    0,0,0,1,SQUARE
        CALL    4,SUM
        STW     0,Y
Z = Y/10 + X/10;
        LW
        L10
        DIV
        LW      0,X
        L10
        DIV
        ADD
        STW     0,Z
```

The HLL of Mini C accelerates programming as any high level programming language does. This means that coding and documenting a program require less time and effort. Maintenance of existing programs becomes easier and changes are quicker to accomplish.

The disadvantages of a HLL for minicomputers are the same disadvantages that have always plagued HLL's. They require compilation and the resultant code executes slower and require additional debugging aids. These shortcomings are minimized for Mini C because the machine conforms to the language. Mini C might properly be called a language oriented machine.

4. COUNTERPOINT 1975 [15]

4.1 The Multilevel Mini

Large scale general purpose maxicomputers may be called Renaissance Computers. A Renaissance Computer is everything to everyone; a multiple purpose, versatile information processor. It provides time-sharing, batch, real-time, and shared data base functions to a diverse community of users. Unfortunately, most Renaissance Computers do not provide all these functions at a cost-effective level. [15, p. 55]

Twenty-five years ago computers of any type were large and expensive because hardware technology was costly. Software programmming was a means of changing the system to fit different applications. The machines were designed to be general purpose and programmable so that many users could share the high cost. Indeed, they had to be Renaissance Computers [15].

It is no longer necessary to perpetuate the Renaissance Computer concept. The mini attitude is an alternate that is practical because hardware costs have sharply declined in the last ten years and resultant simplification of software has partially offset increasing software costs. A mono-computer system implemented on Mini C, for example, can reduce the per capita cost of hardware and software to a point below the per capita costs of a time-sharing system. Additional users merely get replicas of the Mini C system. Integrated data base systems are an exception requiring multiprogrammed access to many users. Even when mini systems are designed with multiple access in mind, they become attractive economically because of their special-purpose nature [17,18].

Simplified software is a result of the minicomputer attitude. Low cost hardware coupled with simplified software opens the door to special purpose computing [18]. Software in a special purpose system is not used to gain a general user base. Instead, software, firmware, and hardware become a continuum of system implementation levels. Each level offers cost/benefit ratios according to the amount of complexity needed at each level. Hardware binds a portion of the complexity so that firmware becomes attractive. Firmware binds complexity at the next level and finally software binds complexity at the application level.

Special purpose, microprogrammable minicomputers of type "Mini C" offer flexibility in terms of bound complexity. Most likely, a blend of "soft-firm-hardware" will prevail in minicomputers of the near future because the benefits offered by multi-level systems are attractive for special purpose computing [15].

If we extrapolate current trends in minicomputer hardware we would predict that minicomputers would soon be free. This, however, is implausible. Instead, a plateau will most likely be reached (at several hundred dollars) at which the price will remain stable. Complexity will be substituted in place of decreasing costs. In the mid 70's we are experiencing the first signs of reaching this plateau [15].

Assuming that multi-level, low-cost minis are close to the price plateau, then we can extrapolate some future directions for minis. Basically, these extrapolations are outgrowths of the need to overcome present limitations. We demonstrated several shortcomings in the previous section. They lead us to believe that the future minicomputer will possess features found on large machines.

(1) More powerful instructions reminiscent of Mini C [9].
(2) Larger word size to gain addressability; hence larger main memories [1].
(3) Architectural extension (ability of user to add instructions to the basic set) through microprogramming [1,7,10].

(4) Special purpose systems through multilevel design (soft-firm-hardware) [9,17,18].
(5) Reduction in size and cost of peripherals used in minicomputer systems [3,10,15].

Where does this leave the large scale computers? Obviously there are many applications that sensibly require the size and power of a large computer. There may be a trend away from Renaissance Computers, but this does not preclude the need for bulk storage capacity and complexity beyond that of a mini.

Large computers in the minicomputer era will probably take advantage of the minicomputer attitude. The large computer of the future might be a distributed network of special purpose (mini) computers [14]. Each subprocessor is a special purpose organ that performs a dedicated function. For example, a storage-control computer is dedicated to managing data into and out of a main memory module. An I/O processor can handle all I/O, an arithmetic processor can perform all arithmetic and a language translator processor can perform only translation.

Minicomputers will continue to compete with larger machines for economic reasons. A buyer in the process of selecting a computer must consider the application, volume of data, and software requirements before choosing between mini and maxi.

4.2 Summary

Minis are finding applications in places where maxis fear to tread. The low cost of hardware leads to specialization, but we must beware of the relatively high costs in programming. In the future, mini hardware will become more complex instead of forever decreasing in cost. Complexity will be distributed across three levels: hardware, firmware, and software [9,10]. High level languages will contribute to decreasing programming costs [7,9,11,12,13]. Systems will be developed on larger computers and loaded into small computers as a part of a mini-maxi symbiosis. Finally, the obvious trend is toward applications and tailored computing [17].

5. REFERENCES

[1] Technical Publications, Varian Data Machines, 2722 Michelson Dr., Irvine, California, 92664.
(a) Software Handbook (98 A 9952 201)
(b) Varian 73 System Processor Manual (98 A 9906 021)
(c) Vortex Reference Manual (98 A 9952 102)
(d) Varian Microprogramming Guide (98 A 9906 072)

[2] International Business Machines, System 370/145 reference manual, DPD Program Information Dept., 40 Saw Mill River Rd.,

Hawthorne, New York 10532.

[3] Distribution, General Automation, 1055 South East St., Anaheim, California 92805.
(a) The Value of Power (89A00064A-D)
(b) The Value of Micro Power (89A00 124A-A)

[4] Publications, Microdata Corp., 17481 Red Hill Ave., Irvine, California 92705.
(a) Computer Reference Manual-Micro 32/S (RM 20003250)
(b) Microdata 32/S Programming Language Reference Manual (MPL)

[5] Software Distribution Center, Digital Equipment Corp., 146 Main St., Maynard, Mass. 01754.
(a) PDP-11 Processor Handbook (1973)
(b) PDP-11 Handbook (1969)
(c) Getting Started with RT-11 (DEC-11-ORCPA-D-D)
(d) RT-11 System Reference Manual (DEC-11-ORUGA-B-D)

[6] Technical Publications, Lockheed Electronics, Sunnyvale, California 95200.
(a) SUE Processor Reference Manual

[7] Manager, Technical Publications, Hewlett-Packard, Data Systems Development Div., 11000 Wolfe Rd., Cupertino, California, 95014.
(a) HP 3000 Reference Manual (03000-900019)
(b) HP 3000 SPL Textbook (03000-90003)
(c) HP 3000 SPL Manual (03000-90002)

[8] Technical Publications, Data General Corp., Southboro, Mass. 01772.
(a) How To Use The NOVA Computers
(b) Introduction To RDOS (093-000083-00)

[9] Burns, R. and D. Savitt, Microprogramming, Stack Architecture Ease Minicomputer Programmer's Burden, Electronics, Feb. 15, 1973, 95-101.

[10] Roberts, W., Microprogramming Concepts and Advantages as Applied to Small Digital Computers, Computer Design, Nov. 1969, 147-150.

[11] Richards, M., BCPL: A Tool for Compiler Writing and System Programming. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J. 557-566.

[12] Ritchie, D. M., C Reference Manual, Bell Telephone Laboratories, Murray Hill, N.J. 07974.

[13] Software Documentation Distribution INTEL Corporation, A Guide To PL/M Programming, Bowers Rd., Santa Clara, California.

[14] DATAMATION, Vol. 21, No. 2, issue on networked minis. Feb. 1975. The advertisements on Page 1, 5, and 7 say a lot about the future of dispersed systems.

[15] Withington, F. G., Beyond 1984: A Technology Forecast, Datamation, Vol. 21, No. 1, Jan. 1975, 54-73.

[16] Lewis, T. G., and Doerr, J. W., Minicomputer Programming Fundamentals, Spartan Book, Hayden Publ. Co., New Rochelle Park, New Jersey, 1976.

[17] Marienthal, L. B., Small Computers for Small Businesses, Datamation, Vol. 21, No. 6, June 1975, P. 62.

[18] Horn, B. K. P., and Winston, P. H., Personal Computers (who needs timesharing?), Datamation, Vol. 21, No. 5, May 1975, p. 111.

NBS-114A (REV. 7-73)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. NBS IR 76-1033	2. Gov't Accession No.	3. Recipient's Accession No.

4. TITLE AND SUBTITLE

Minicomputers: An Attitude

5. Publication Date: March 1976

6. Performing Organization Code

7. AUTHOR(S)

T. G. Lewis

8. Performing Organ. Report No.

9. PERFORMING ORGANIZATION NAME AND ADDRESS

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

10. Project/Task/Work Unit No. 640-1129

11. Contract/Grant No.

12. Sponsoring Organization Name and Complete Address *(Street, City, State, ZIP)*

National Bureau of Standards &
National Science Foundation
Washington, D. C.

13. Type of Report & Period Covered: Final Report

14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

16. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)*

Minicomputers are defined in dozens of ways: by word length, memory size, speed, cost, applications, peripherals, software, and design. The definition used here includes all limited resource computers and more importantly emphasizes the attitude behind minicomputing, that the undertaking be frugal but adequate. Minicomputer architectures are categorized according to type of control (random logic or microprogrammed), bus structure (distributed or central), number of working registers, and instructions (special purpose and limited). The minicomputer attitude is defined as the attitude that places importance upon design of simple, straightforward, special purpose, dedicated computing systems. Programming and software is the most significant problem faced within minicomputing. This suggests that complexity should be forced into hardware because hardware is less expensive than software. High level language support is needed.

Three demonstration minis are used to show how hardware complexity influences software complexity (and therefore software cost).

Concluding speculations suggest minis will overcome current limitations, will incorporate more complexity into hardware, and use the multi-level nature of soft-, firm-, and hardware to advantage in developing special purpose systems.

17. KEY WORDS *(six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons)* Architecture; assembly language; LSI; microprogramming; minicomputer; physical I/O; programming techniques for small computers; stack processing.

18. AVAILABILITY

☐ Unlimited

☒ For Official Distribution. Do Not Release to NTIS

☐ Order From Sup. of Doc., U.S. Government Printing Office Washington, D.C. 20402, SD Cat. No. C13

☐ Order From National Technical Information Service (NTIS) Springfield, Virginia 22151

19. SECURITY CLASS (THIS REPORT): UNCLASSIFIED

21. NO. OF PAGES: 50

20. SECURITY CLASS (THIS PAGE): UNCLASSIFIED

22. Price

USCOMM-DC 29042-P74